Socket Assignment Report

Partners: Nihal Taşcı (2264687), Tahira Kazimi (2415354)

In this assignment, we implemented two file transfer protocols, namely TCP and UDP for transferring 20 objects of varying sizes. The codes in our implementations indicates that for TCP we used an acknowledgement mechanism in both client and server sides, and sent the data by adding a terminating null character at the end. Additionally, adding a variable sleep time helped in not over-crowding the socket. Without the sleep and acknowledgement messages, we found out that TCP is not sending the packets and the connection closes prematurely.

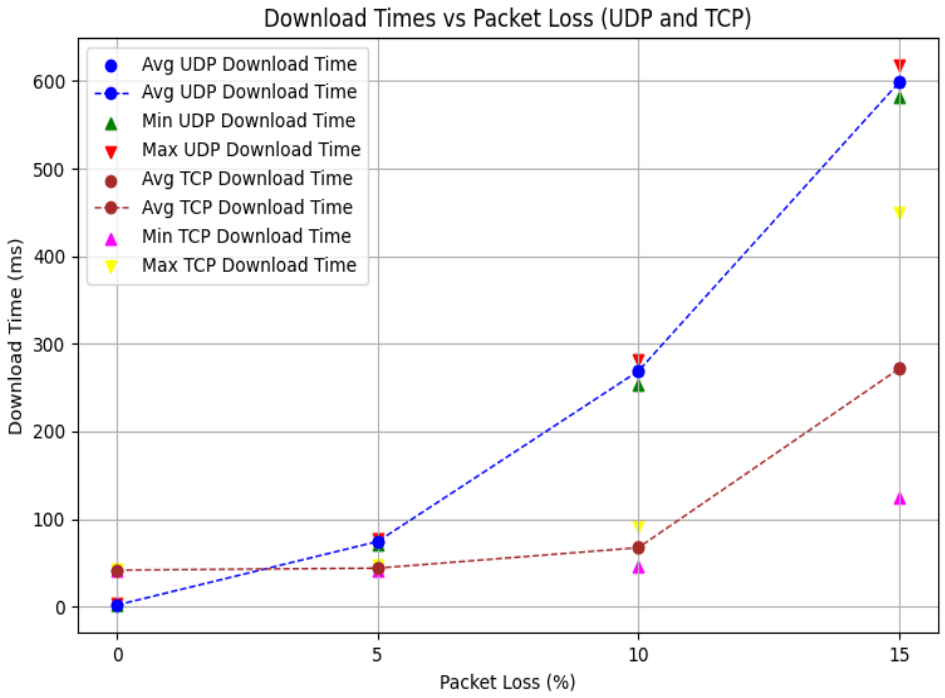
On the other hand, for UDP file transfer, which is an unreliable data transfer, we implemented a pipelining mechanism to expedite the packet sending process and avoid head of the line blocking. The approach we conducted is Go Back N. This approach uses a window size to send all packets at once, and wait for their acknowledgement or resend the packets in case of a timeout. In the sender side, client keeps sending the packets in a window, then it sets a timeout for how long it waits on the receiving side for an ACK message. If a timeout occurs, the sender breaks out of waiting loop and resends the same window. If the ACK for all of the packets sent in window is received, the window moves forward until all data is sent, after which the client sends a FIN message, indicating to close the connection. In the server side, the server continuously waits for a packet with an expected sequence number, if the expected sequence number is received, it unpacks the data, otherwise discards it. In the case that server receives an already ACKed packet, it sends an ACK to client. If the sequence number is out of the window range, it simply discards it and wait on the socket. Finally, if a FIN message is received, it sends a FIN_ACK message back and closes the connection. In case of packet loss, the client sends 3 FIN message and terminates the connection.

We ran multiple experiments over different Netem rules for both of out implemented TCP and UDP. The figures and explanations are as follows:

1. Packet Loss (0%, 5%, 10%, 15%):

   The graph below shows the average, minimum, and maximum values for each packet loss percentage over both TCP and UDP. As seen in the figure, the packet loss did not greatly affect the performance of TCP. TCP protocol seems to stay around the same values for most packet loss metrics. For example, without any Netem rules applied, the TCP download time of objects along with their checksums is around ~40 seconds. As the packet loss increases, the TCP shows more consistency and steadily increases to intervals of 60 and 100 seconds. This is because of all the congestion control mechanisms applied over TCP protocol. It not only does not overflow the link but also ensures a reliable data transfer, which makes it more adaptable to severe network conditions. On the other hand, our implemented UDP uses a sliding window with GBN mechanism. After many trial and errors, we found out that GBN's performance under different packet loss conditions is directly affected by sender's

window size and the timeout value in which sender waits for ACK messages. By adjusting different window size and timeout value, we realized that the window size of 25 and timeout of 0.7 in the client side sends the data in the most optimal way with minimum retransmission effect.



According to our runs, the UDP protocol with GBN performs better or close to TCP for packet losses 0 and 5, the problem exacerbates in UDP as the packet loss gets worse. This could be caused by the fact that GBN sends all the packets in a window for a single packet loss.

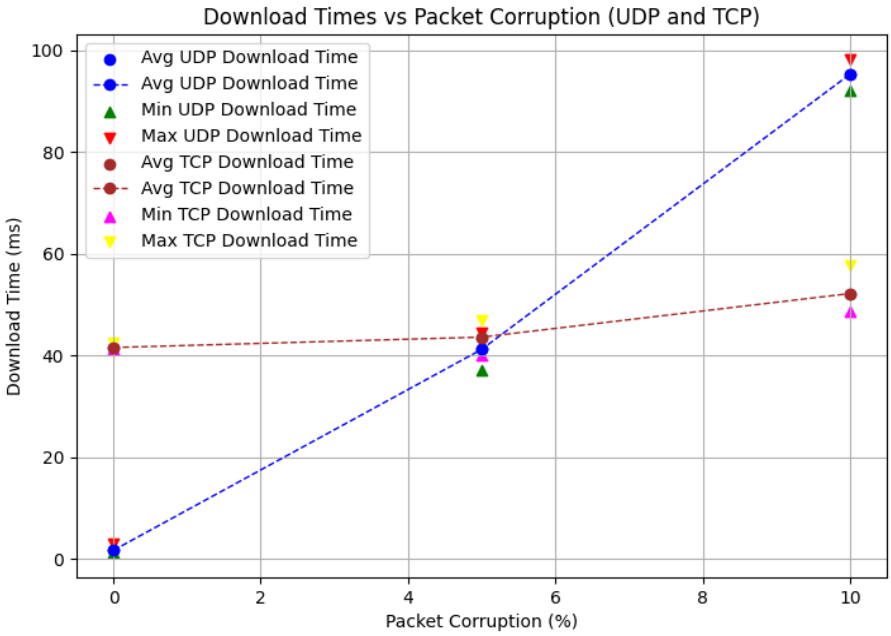
The average download time for packet loss 10% for UDP and TCP are 268 and 67, respectively, with worst overhead of 200 seconds.

The average download time for packet loss 15% for UDP and TCP are 598 and 271, respectively, with worst overhead of 327 seconds.

2. Packet Corruption (0%, 5%, 10%):

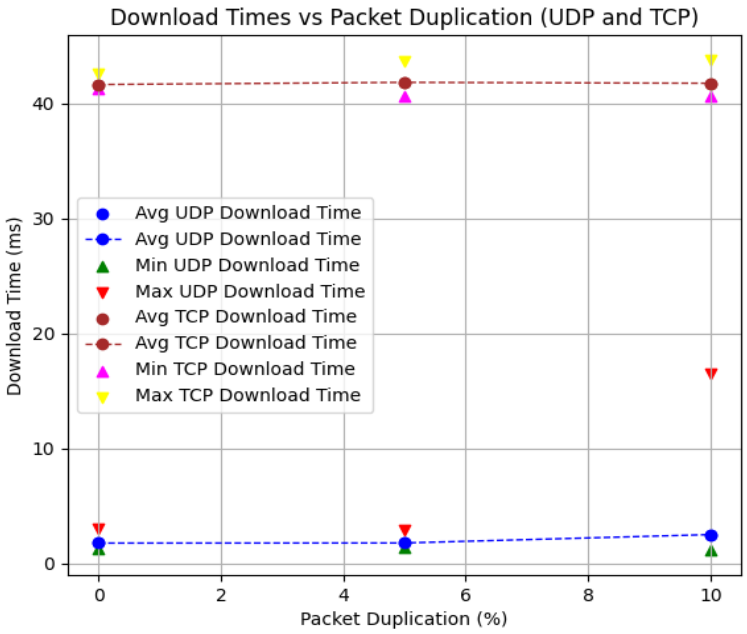
In case of packet corruption, again, UDP performs better or in lines with TCP for packet corruptions of 0% and 5%. Because of the intrinsic behavior of UDP that overflows the link with packets in the same window size, the corruption performance diverges from the TCP performance in 10% corruption rate. Also, we believe that in the UDP the server side sends ACK messages for already acked packets could cause extra traffic in the link. Unfortunately, if we don't send ACKs from the server side for duplicate packets, the client may enter a timeout, and then resend the whole window again. So there is a trade-off in both cases. We found the former approach to be more practical.

In any case, the overhead of Go Back N compared to TCP is around 40 seconds. For 10% corruption rate, UDP average download time is 95, whereas for TCP it is 52 seconds.



3. Packet Duplication (0%, 5%, 10%):

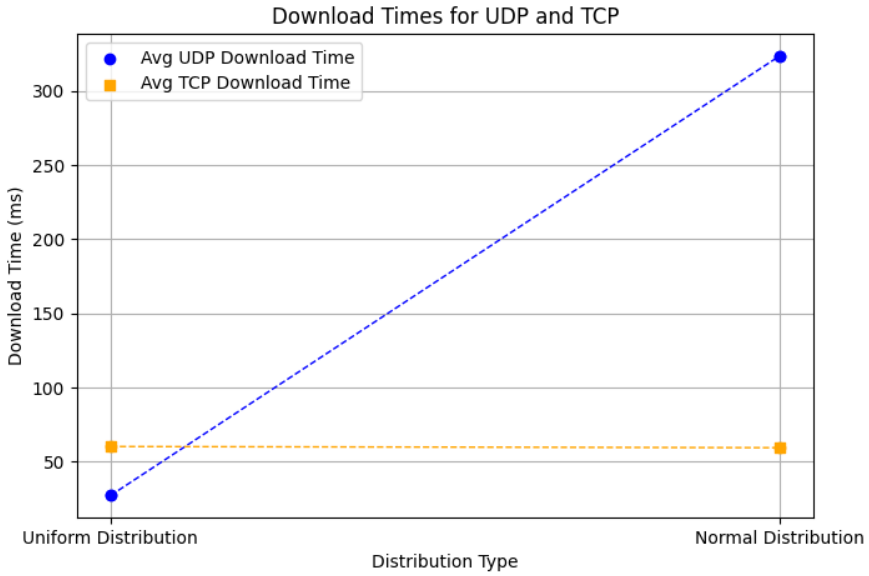
For packet duplication, our UDP performs considerably better than the TCP, with averages 1.75, 1.76, 2.49 for 0%, 5%, and 10% respectively. On the other hand, the TCP tends to stay around the same benchmark 0 netem values of average 41 seconds for all 3 cases. As packet duplication doesn't really affect the link's congestion rate and it only increases the number of packets to send through the link, under normal conditions, UDP can overflow the link which will result in faster performance compared to its TCP counterpart. TCP, on the other hand follows a steady rate, with a slow start, which preserves the link's traffic rate.



4. Normal distribution vs Uniform distribution:

For Uniform Distribution, the packets experience a constant delay in both sides, this behavior will cause the UDP to add some constant delay to its packet transfer. However, adding a 100ms delay still performs relatively better than TCP. By increasing the delay value the uniform distribution may exhibit longer download periods than TCP. Normal distribution of 100ms delay with 20ms standard deviation causes UDP to have a sudden increase in download time. This happens mostly because of varying time delay in sending packets and receiving ACK messages in the client side, which results in frequent timeouts and retransmissions.

Since TCP uses a dynamically adjustable flow control mechanism, the 100ms delay does not affect the download time severely. The flow control system adapted by TCP enables this protocol to send packets based on the link condition and dynamically adjust the sending rate.



Conclusion:

In short, our experiment reveals that pipelined UDP can significantly outperform TCP in cases with no or minimal traffic on the link, due to no congestion control. However, because the pipelined UDP only tries to optimize the link utilization, and overflows the link, it performs worse when the congestion increases in the link. We adapted a Go Back N approach to maximize link utilization. However alternative approaches can be used such as Selective Repeat. We believe that Selective Repeat could potentially perform better than GBN in congested situations, though its performance cannot potentially exceed the TCP due to the nature of pipelining algorithm of sliding window. In other words, the main objective behind SR is also maximizing the link utilization. It does not take into account the congestion control mechanisms or care about the control flow, as to not overcrowd the routers and links. As a result, if a link is inherently congested, the SR also reaches a bottleneck in which it cannot perform better than TCP without implementing a dynamic congestion control mechanism. One way to improve this pipelined UDP bottleneck is to communicate in low level with network routers explicitely, which can set some bits in transport layer headers. This will inform the UDP protocol layer about the state of connecting routers. As a result, both SR and GBN will have a performance bottleneck compared to TCP protocol in congested links.